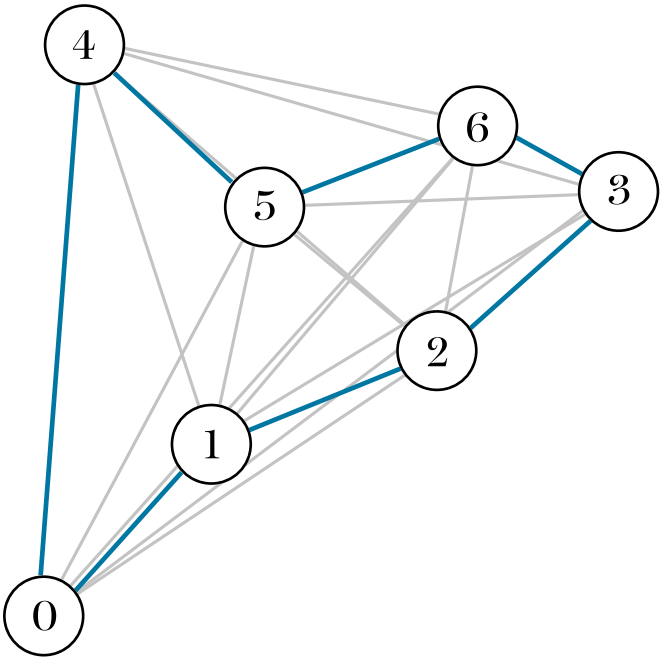
TP2: Reducing the costs

Victor Pires Diniz

14 de Junho de 2015

Algoritmos e Estruturas de Dados III - 1º Semestre de 2015



1 Introdução

A computação facilita a solução de vários problemas por meio de algoritmos. A ideia é que esses algoritmos sejam capazes de explorar propriedades do problema para que se possa encontrar, eficientemente, a solução. No entanto, há problemas para os quais até hoje não foi possível encontrar procedimentos eficazes, devido à falta de propriedades úteis evidentes. Em particular, a classe de problemas NP-completo contém problemas para os quais grande parte da comunidade acredita não haver solução polinomial. Isso significa que é necessário optar por uma solução sub-ótima ou lidar com algoritmos exponenciais, que rapidamente se tornam inviáveis conforme cresce o tamanho da entrada.

Neste trabalho, o cenário envolve uma empresa de transporte, que deve, partindo de uma cidade inicial, visitar todas as cidades e retornar ao ponto de partida no menor tempo possível. Além disso, há um conjunto de restrições de ordem de visita, que podem mudar a solução obtida ou, até mesmo, inviabilizar a existência de uma solução. Esse problema é uma variação de um problema notoriamente difícil. Posteriormente, será provado que não se conhece um algoritmo polinomial para esse problema, visto que sua versão de decisão é NP-Completa.

É necessário obter a solução ótima, o que significa que não é possível evitar algoritmos que operem em tempo exponencial. No entanto, isso não quer dizer que não haja maneiras de otimizar a solução. Podas e abordagens de *branch-and-bound* são formas de limitar o espaço de busca, tornando mais eficiente a enumeração realizada e permitindo entradas um pouco maiores. Por mais que, eventualmente, uma entrada vá

se tornar inviável, esse tipo de melhoria pode ser suficiente para situações reais. No caso deste trabalho, os aprimoramentos realizados fizeram com que entradas anteriormente maiores do que o programa era capaz de resolver se tornassem solucionáveis em tempo aceitável.

2 Formalização do problema e NP-Completude

O problema apresentado neste trabalho é uma variante de um dos problemas mais famosos na história da ciência da computação: o Problema do Caixeiro Viajante (PCV). Esse problema envolve descobrir o percurso ótimo, partindo de uma cidade arbitrária, de forma a visitar todas as outras cidades e retornar ao ponto inicial. Neste trabalho, há um fator adicional a ser considerado: existe um conjunto de restrições de ordem de acesso.

2.1 Modelagem em grafos

A entrada do problema tratado neste trabalho é composta por um conjunto de n cidades e um conjunto de restrições. No entanto, essas cidades são representadas como pares ordenados que denotam sua posição em relação a um ponto arbitrário. Para facilitar a representação e resolução do problema, é feita uma modelagem em grafos. O grafo utilizado é um grafo não-direcionado completo G, de n vértices, no qual cada aresta representa a distância euclidiana entre as duas cidades conectadas.

2.2 Formalização como problema de decisão

Por ser um problema de otimização, o cenário da empresa de transportes apresentado neste trabalho não é NP-completo, mas sim NP-difícil. Isso ocorre porque, nos problemas NP-completo, é necessário que seja possível verificar a saída do problema em tempo polinomial. Os problemas de otimização requerem que se obtenha o melhor resultado possível, e verificar se um resultado é o melhor possível consiste em resolver o problema novamente (assume-se que isso é impossível em tempo polinomial para problemas NP-difícil). No entanto, é possível formalizar o problema apresentado como um problema de decisão e, então, provar que essa versão dele é um problema NP-completo. Uma forma de realizar tal reformulação é a seguinte:

"Dados uma lista de n cidades, as distâncias entre essas cidades, um limite superior $\in \mathbb{R}_+$ e um conjunto de restrições de ordem de acesso, existe uma rota que visita todas as cidades exatamente uma vez e, então, retorna à cidade original sem violar nenhuma das restrições propostas, cuja distância é menor que k?"

2.2.1 Configurações triviais do problema de decisão

Essa versão de decisão do problema permite que, para algumas configurações particulares, a resposta seja obtida trivialmente. Caso as restrições formem ciclos, é impossível que haja circuitos válidos nesse grafo, visto que as restrições nunca permitirão o acesso a nenhum dos vértices presentes no ciclo. Além disso, o valor assumido por k pode, também, trivializar o problema: caso esse valor seja menor do que a soma das $n+1$ menores arestas, não existe caminho válido. Similarmente, caso k seja maior do que as $k+1$ maiores arestas e haja no máximo uma restrição, é possível dizer com certeza que existe tal rota.

2.3 Prova de NP-completude

2.3.1 Algoritmo de verificação polinomial

Primeiramente, é preciso provar que o problema é **NP**. Isso pode ser feito através da apresentação de um algoritmo de verificação do resultado do problema em tempo polinomial. No caso do problema em questão, isso é bastante simples. Dado o caminho a ser tomado pelo caminhão da empresa de transportes, o algoritmo se resume a seguir o trajeto e garantir que ele não viola nenhum dos critérios. Ele deve ter distância final menor do que k, percorrer todas as cidades, partindo da cidade 0 e voltar à cidade original.

Algorithm 1: Pseudocódigo do algoritmo de verificação

Entrada: Vetor de cidades visitadas V, grafo de cidades e distâncias G, limite superior da distância do trajeto k e conjunto de restrições R.
Saída : Valor booleano *true* se o caminho é válido, *false* caso contrário.

if $V[0] \neq 0$ ***or*** $V[-1] \neq 0$ **then**
  return *false*
end
$C \leftarrow \{0\}$ // Cidades já visitadas
$D \leftarrow 0$ // Distância percorrida
$P \leftarrow 0$ // Cidade anterior
for v *in* $V[1:]$ **do**
  if $v \in C$ ***or*** *Cidades visitadas em* C *não satisfazem as restrições de* $R[v]$ **then**
    return *false*
  else
    $D += G_{P,v}$ // Distância entre P e v
    C.append(v)
    $P \leftarrow v$
  end
end
if $D \leq k$ **then**
  return *true*
else
  return *false*
end

Considerando as operações de checar se uma cidade já foi visitada e a conferência das restrições de acesso como $O(n)$, o algoritmo de verificação apresentado acima tem ordem de complexidade $O(n^2)$. Portanto, o problema em questão é NP.

2.3.2 Prova de NP-dificuldade: redução polinomial

A prova de que o problema proposto no trabalho é NP-difícil pode ser feita através de redução polinomial do Problema do Circuito Hamiltoniano (PCH), um problema NP-completo (e, consequentemente, NP-difícil) conhecido. O enunciado do PCH é: "dado um grafo $G(V,A)$, é possível estabelecer um circuito nesse grafo que visite todos os vértices apenas uma vez e retorne ao vértice original?"

No caso, é preciso transformar a entrada de uma instância genérica do PCH na entrada do PCV com restrições, que é o problema que se quer provar NP-completo. O grafo de entrada do PCV com restrições é um grafo completo, já que existe distância euclidiana entre todas as cidades. A transformação a ser realizada envolve um novo grafo $H(V,A')$, baseado no grafo $G(V,A)$, e funciona da seguinte maneira:

$$A'_{i,j} = \begin{cases} 0, & \text{se existe } A_{i,j}. \\ 1, & \text{caso contrário.} \end{cases}$$

O grafo $H(V,A')$ pode ser gerado de maneira simples em tempo polinomial. O conjunto de restrições R é vazio. Com isso, resta provar que existe uma rota válida para o PCV com restrições com grafo $H(V,A')$ e conjunto de restrições R de tamanho 0 **se e somente se** existe um circuito hamiltoniano em $G(V,A)$.

Se o grafo $G(V,A)$ apresenta um circuito hamiltoniano C, isso significa que cada aresta em C existe em A, e, por isso, tem custo 0 em A'. Por essa razão, C também é uma rota válida em $H(V,A')$ com custo zero. Similarmente, se D é uma rota válida em $H(V,A')$ e tem custo 0, isso significa que todas as suas arestas estão presentes em A (caso contrário, o custo de D seria maior do que zero), e, por isso, D é um circuito hamiltoniano válido em $G(V,A)$.

2.4 Podas utilizadas

Para evitar a enumeração de todas as permutações de vértices possíveis, foram utilizadas propriedades do problema para identificar pontos na árvore de decisão a partir das quais todas as possibilidades são infru-

tíferas. Com isso, é possível interromper imediatamente a recursão e evitar muitos passos desnecessários. Essas podas não trazem um ganho na ordem de complexidade do problema, mas elas permitem que entradas maiores passem a ser tratáveis.

Há dois tipos de podas: podas locais são aquelas que decidem quando interromper a recursão apenas com base na configuração local do problema, enquanto as podas globais utilizam informações obtidas em iterações anteriores para constatar que a solução para o problema não está nesse galho da árvore de recursão, muitas vezes através de variáveis com escopo global.

2.4.1 Unicidade

Essa poda local é simples e vem naturalmente na implementação do algoritmo para a solução do problema. Ela se baseia no fato de que vértices não podem se repetir na rota a ser tomada, o que significa que caminhos que visitam a mesma cidade mais de uma vez não precisam ser explorados. Ela se aplica para todas as configurações do problema, com a exceção única do caso trivial em que só existe uma cidade. Algoritmos que não aplicam esse princípio operam em complexidade de tempo $O(n^n)$, ao invés de $O(n!)$, o que significa que essa poda é de extrema importância.

2.4.2 Restrições

Ao invés de conferir as restrições somente quando é obtido um caminho completo de tamanho $n+1$, é possível ganhar eficiência realizando uma poda local que se baseia na conferência das restrições a cada cidade visitada. Isso faz com que as sub-árvores nas quais restrições são violadas sejam completamente ignoradas. Quanto mais restrições houver, mais eficiente essa poda é. Ela não se aplica no caso em que não há restrições.

2.4.3 Simetria

A poda local de simetria é uma poda para o PCV tradicional em que se aproveita o fato de que uma rota é equivalente à sua rota inversa, e, para obter o resultado ótimo, basta encontrar uma dessas duas rotas. Dessa forma, é imposta uma restrição de ordem de acesso, que necessariamente invalida uma das rotas, mas mantém a outra. No problema deste trabalho, porém, essa poda não é válida caso haja restrições, visto que adicionar uma restrição caso já existam restrições pode fazer com que a rota ótima nunca seja encontrada. Dessa forma, essa poda somente se aplica para o caso em que não há restrições. Apesar da sua limitação, a implementação dessa poda é muito simples (basta adicionar uma restrição arbitrária caso não haja restrições) e ela se aplica justamente no caso de pior performance possível, já que o desempenho do programa é melhor quanto mais restrições há.

2.4.4 Limite superior

A poda global utilizada nesse trabalho é realizada através da manutenção do valor do menor caminho completo encontrado até o momento. Devido ao fato de que não há arestas negativas, se uma rota parcial é maior do que a menor rota completa encontrada até o momento, nenhuma das rotas que podem ser criadas adicionando arestas a essa rota pode ser melhor do que o mínimo encontrado. Por isso, é possível descartar a sub-árvore encontrada até esse ponto.

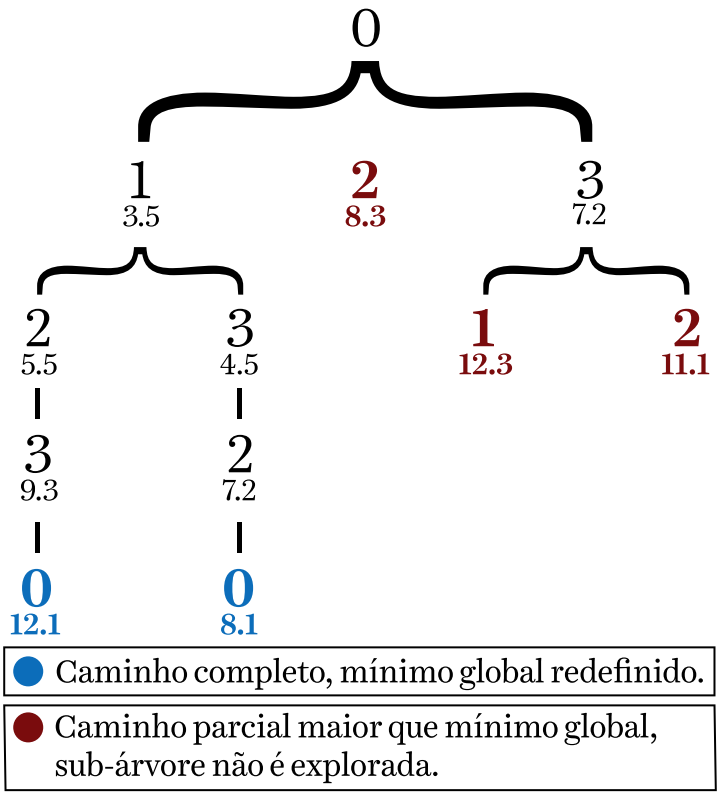


Figura 1: Árvore de enumeração de possibilidades para o caso de quatro vértices, sem restrições, demonstrando a podas de limite superior. Leitura da esquerda para a direita.

A eficácia dessa poda depende da configuração do grafo de distâncias e na ordem de enumeração dos vértices. O desempenho da poda diminui conforme a configuração das rotas completas se aproxima da ordem decrescente e do final da árvore de enumeração. Por outro lado, caso uma rota curta seja encontrada cedo no procedimento, é possível economizar muitas iterações. Essa poda não se aplica caso não haja rotas válidas.

3 Implementação

3.1 Grafo como matriz de adjacência (*amgraph.c*)

A biblioteca *amgraph.c* é uma implementação simples de um grafo não direcionado como uma matriz de adjacência. A função *amgInit* é responsável pela inicialização do grafo como um grafo vazio de n vértices. Arestas podem ser adicionadas através do procedimento *amgSetEdge*, e seus pesos podem ser conferidos por meio de *amgGetWeight*. Por fim, *amgDelete* libera a memória alocada na inicialização do grafo.

A escolha pela matriz de adjacência se deu pelo fato de que operações de obtenção do peso das arestas do grafo são extremamente frequentes no procedimento recursivo da busca exaustiva. O grafo pode ter no máximo 22 vértices, pelas limitações propostas pelo problema, o que significa que o gasto de memória elevado dessa representação é praticamente insignificante.

3.2 Solução do problema por busca exaustiva (*tsp.c*)

tsp.c é a biblioteca com a funcionalidade principal do programa: resolver o PCV com as restrições impostas. Apesar de evitar a exponencialidade do problema ser assumidamente impossível, técnicas para atenuar a degradação do desempenho do programa são aplicadas, de forma a viabilizar a solução do problema para números um pouco maiores de cidades.

A função *buildDistanceGraph* cria e povoa o grafo cujas arestas são a distância entre as cidades, com base nas coordenadas de cada cidade. Para isso, é utilizada a função auxiliar *dist*, que simplesmente calcula a distância euclidiana entre dois pontos.

findMinDistance é uma função que serve basicamente para chamar as outras funções da biblioteca e dar início ao processo recursivo de enumeração. No entanto, a função que realiza, de fato, a busca exaustiva é *recursiveCall*. Essa função mantém informações em escopo local sobre os vértices que já foram visitados, o vértice atual e a distância percorrida, além de informação global sobre a menor distância encontrada até então.

Function recursiveCall

Data: Grafo de cidades e distâncias G, conjunto de cidades visitadas V, cidade atual C, distância percorrida d, conjunto de restrições R, e mínimo global D.
Result: Ao final, D contém o valor do menor trajeto encontrado. Caso esse valor não tenha sido definido, não foi possível encontrar um caminho válido, dadas as restrições impostas.

```
V.insert(C)
if V contém todas as cidades then
    d += G_{C,0}                    // Soma distância para retornar à cidade de origem
    if d < D then
        D ← d
    end
    return
end
for Cidade c ∉ V do
    if V satisfaz R_c then
        d' ← d + G_{C,c}
        if d' ≤ D then
            recursiveCall(G, V, c, d', R, D)
        end
    end
end
```

A cada chamada, primeiramente é checado se todos os vértices já foram visitados. Nesse caso, soma-se a distância do ponto atual à origem e, se essa distância for menor do que o melhor caminho encontrado até o momento, o novo trajeto é considerado o melhor.

Caso contrário, o loop presente na função tenta, como próximo destino, todas as cidades que não foram visitadas ainda e não tem acesso restrito, considerando as cidades visitadas até o momento. É conferido, para cada cidade, se a adição de tal cidade ao caminho fará com que o caminho em questão ultrapasse o mínimo global encontrado até o momento. A ocorrência disso revela a infrutiferidade de toda a sub-árvore de decisão desse segmento no que cabe a encontrar o melhor caminho.

É possível que não haja caminhos válidos para um conjunto de restrições, o que é considerado um *deadlock*. Isso é avaliado através da atribuição inicial da distância mínima como infinito. Caso nenhum caminho tenha sido encontrado ao final do procedimento, esse valor continuará sendo infinito, e, com isso, é possível constatar o *deadlock*.

Um detalhe relevante da implementação é que, devido ao limite superior de 22 cidades, foi possível avaliar as restrições para um determinado estado em tempo constante, através de máscaras de bits e operações *bitwise*. Ao invés de guardar os vértices já visitados em um vetor, é possível fazer isso em um simples inteiro, considerando o bit menos significativo como o estado da cidade #0, o segundo menos significativo com o da cidade #1 e assim por diante. As restrições foram representadas também dessa forma: se o terceiro bit mais significativo da restrição da cidade #3 é 1, isso significa que é preciso visitar a cidade #2 antes da #3, por exemplo. A conferência das restrições, então, é feita com um *bitwise and* entre a máscara de cidades visitadas e as restrições. Se esse operador retorna o mesmo valor da máscara de restrições, isso significa que todas as restrições foram correspondidas com sucesso.

3.3 Arquivo principal (*main.c*)

Este arquivo contém apenas a função *main*. Ele analisa a entrada de cada caso de teste e constrói as máscaras de restrição e os vetores de coordenadas das cidades. Então, ele chama a função *findMinDistance*, explicada anteriormente, e imprime o resultado da computação realizada.

4 Análise de complexidade assintótica

4.1 *amgraph.c*

A análise de complexidade assintótica da biblioteca responsável pela representação de grafos como matrizes de adjacência é realizada em função do número de vértices, V.

amgInit

Esta função simplesmente inicializa o tipo abstrato de dados, alocando a memória necessária e realizando um número constante de atribuições. Sua complexidade temporal é $O(1)$, visto que a função *malloc* não tem seu desempenho como função do tamanho de memória a ser alocado, mas sim da configuração das alocações realizadas, entre outros fatores não relacionados. Por outro lado, esta função tem complexidade de espaço $O(V^2)$, já que é alocada uma matriz quadrada de lado V.

Outras funções

amgSetEdge e *amgGetWeight* são apenas funções de interface, que operam em tempo constante. *amgDelete* é responsável pela liberação da memória da estrutura, que também é uma operação de desempenho independente do tamanho do grafo.

4.2 *tsp.c*

A análise de complexidade assintótica da biblioteca responsável pela solução do problema é realizada em função do número de cidades, n.

4.2.1 *buildDistanceGraph*

Esta função é responsável pela criação do grafo de distâncias com base no vetor de coordenadas. O número de arestas do grafo equivale a $C_{n,2}$.

$$O(C_{n,2}) = O(\frac{n*(n-1)}{2}) = O(n^2) \tag{4.1}$$

Sua complexidade de espaço corresponde ao gasto da função *amgInit*. Por isso, essa função tem complexidade espacial $O(n^2)$.

4.2.2 *recursiveCall*

Nesta função, o fator mais relevante para a análise de complexidade assintótica é a chamada recursiva realizada. É possível expressar essa relação como uma relação de recorrência. Considerando r como o número de cidades restantes a serem visitadas, esta função tem sua recorrência representada pela função $T(r)$, onde:

$$T(r) = \begin{cases} 0, & \text{se } r = 0. \\ r\,T(r-1), & \text{caso contrário.} \end{cases} \tag{4.2}$$

A expansão em termos dessa recorrência resulta em uma ordem de complexidade de tempo fatorial $O(n!)$.

A complexidade de espaço, por outro lado, é linear, já que a profundidade recursiva em um determinado instante corresponde ao número de cidades visitadas nesse momento.

4.2.3 *findMinDistance*

Essa função tem sua complexidade de tempo definida pela chamada a *recursiveCall* e, por isso, é $O(n!)$. Quanto à complexidade espacial, por outro lado, predomina *buildDistanceGraph*, com complexidade quadrática $O(n^2)$.

4.3 Programa principal (*main.c*)

O programa principal apenas trata entrada e saída, gera as restrições e chama a função *findMinDistance*. Por essa razão, predomina a complexidade dessa função, tanto em tempo quanto em espaço.

5 Análise experimental

Os testes realizados analisaram o fator mais relevante e mensurável para o desempenho do programa: o número de cidades da entrada. Há, também, outros fatores que podem influenciar no desempenho, como, por exemplo, as restrições impostas. No entanto, esses fatores são mais difíceis de avaliar, visto que eles não dependem simplesmente de um valor numérico. No caso das restrições, não só o número de restrições afeta o tempo de execução do programa, mas também a configuração das mesmas. Restrições simétricas, por exemplo, podem causar grande redução no tempo de execução do programa, já que isso caracteriza um *deadlock*. Por outro lado, um grande número de restrições partindo de um mesmo vértice podem ter impacto muito menor. Isso também depende da ordem de avaliação dos vértices no programa, que é completamente arbitrária.

Todos os testes foram realizados dez vezes cada em uma máquina com as seguintes especificações:

- OS: Elementary OS x64 (distribuição baseada no Ubuntu 14.04 LTS);
- CPU: Intel Core i7 2600k @ 3.40GHz;
- RAM: 4 x 4GB DDR3 1.333MHz (16GB).

A medição dos tempos de execução foi feita através do comando *time*, no terminal. Todos os testes foram realizados no dia 14 de junho de 2015.

5.1 Resultados

Nesta bateria de testes, o programa recebeu como entrada para cada caso de testes um conjunto de n cidades com coordenadas aleatórias inteiras de 0 a 10^9, sem nenhuma restrição. Houve casos de testes com 10, 11, 12, 13, 14, 15, 16 e 17 cidades. Os resultados estão disponíveis no gráfico abaixo:

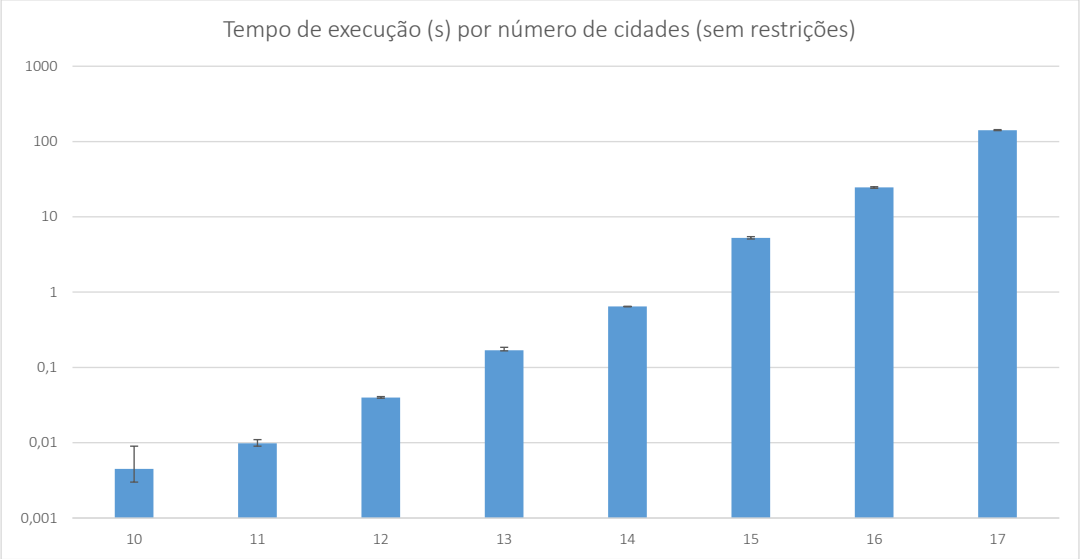


Figura 2: Gráfico do tempo de execução dos testes (em segundos) em função do número de cidades, em escala logarítmica (base 10).

O comportamento desse teste está de acordo com a ordem de complexidade do programa. Na seção de análise de complexidade assintótica, constatou-se o comportamento exponencial do tempo de execução do programa conforme o número de cidades aumenta. A linearidade aparente do gráfico em escala exponencial confirma, de fato, a exponencialidade em função do número de cidades da complexidade de tempo do programa. As barras de erro mostram o maior e o menor tempo de execução obtidos para cada caso de teste.

6 Conclusão

Nesse trabalho, foi abordado um problema notoriamente difícil. Como manter a otimalidade era necessário, a solução proposta, apesar de todas as otimizações realizadas, não é capaz de tratar entradas grandes. Foi provado, porém, que se trata de um problema NP-completo, e, por isso, acredita-se não ser possível obter um algoritmo polinomial que preserve a otimalidade das soluções obtidas. Podas e otimizações foram realizadas para tratar entradas um pouco maiores, de forma a cumprir os requisitos máximos propostos pelo enunciado do trabalho. Também foram feitas análises de complexidade assintótica e experimental, que revelaram, novamente, a exponencialidade do problema.

Existem algoritmos que são capazes de resolver o problema para entradas ainda maiores. A otimização de problemas NP-completos é um tema muito frequentemente abordado na ciência da computação. Existem relatos de soluções capazes de tratar entradas que alcançam as dezenas de milhares de vértices. No entanto, tais métodos fogem do escopo do trabalho e da matéria abordada, além de serem desnecessárias para a entrada máxima necessária, de 22 vértices.